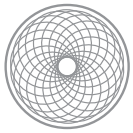


Mini-Projet PASD

Rapport de Projet PASD

Préparé pour: Jérôme DURAND-LOSE, Titre : compte-rendu

Préparé par:

Soufiane Boulealf

Hubert Simon

Aflah Sofiane

Rédaction :

- Structure des files : Hubert Simon
- Sort, Gestion de Mémoire, Option ligne commande, Flux, Test : Boulealf Soufiane
- Option ligne de taille quelconque : Aflah Sofiane

22 octobre 2012

Tables des Matières

Gestion de Mémoire

Avant Même de commencer à coder nous devons nous fixer des règles :

- Aucun programme ne doit allouer de la mémoire à part le programme de gestion de lecture «read_alloc», le programme de sort.
- Il faut utiliser le free dans le même corps, où si je peux le dire, dans le même niveau que malloc. pour éviter de supprimer les éléments qui n'ont pas été alloué, car on a pas pu accéder à la condition ...

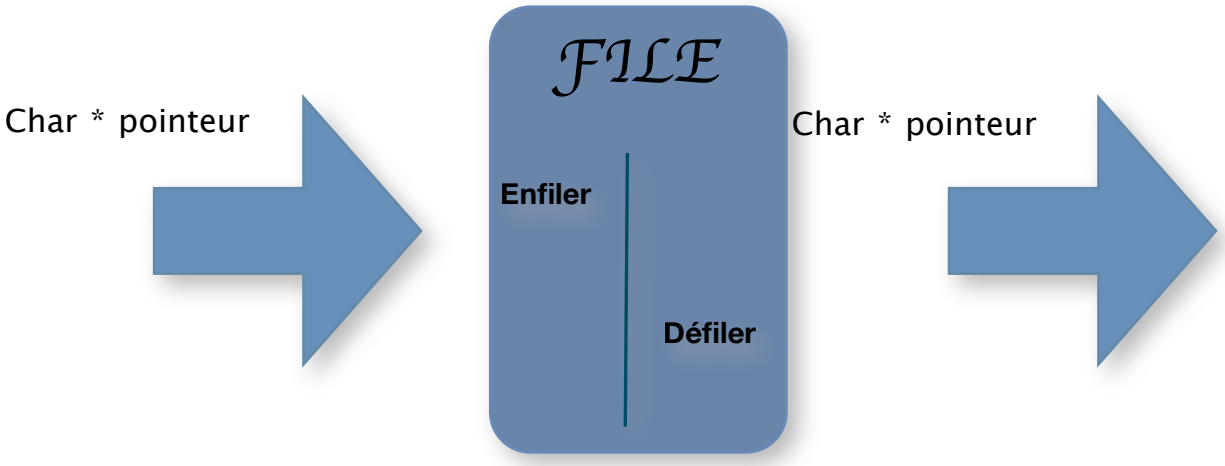
  Exemple :

```
if (conditon 1){
        var = malloc(1)
        ...
        ...
}
free (var)
```

- La fonction défiler doit retourner impérativement l'élément en question, nous allons éviter de manipuler les maillons au niveau des entrées et sorties des sous-programmes, donc on va retourner la chaine de caractère contenu dans le maillon en question, ce qui implique un free du Maillon défilé, et un malloc du pointeur de chaine de caractère retourné.

  Ce dernier va être soit transféré dans une autre file, (fonction transférer), ou supprimé manuellement avec la fonction «write_remove», qui vide la file dans la sortie standard, tout en supprimant chaque élément après utilisation.

Modélisation :



Gestion de flux entrée/sortie

Lecture et allocation mémoire « read_alloc »

Une fonction « read_alloc » a été définie afin de s'occuper de l'enregistrement des informations récupérées au niveau de l'entrée standard (qui peut être un fichier c.f extensions).

Entrées :

- La file réceptrice
- Le pointeur de fichier qui est sensé pointer vers l'entrée standard ou un fichier (c.f extension).

Cette fonction alloue de la mémoire à chaque fois que l'entrée standard envoi des informations, jusqu'à fermeture du flux d'entrée. Ensuite enfile ses informations.
N.B : Sans oublier de supprimer la dernière allocation de mémoire faites pour le pointeur de caractère.

Écriture et suppression des données « write_remove »

Une fonction « write_remove » a été définie pour «balancer» les données sur la sortie standard par défaut, ou un fichier en cas de remplissage

Entrées :

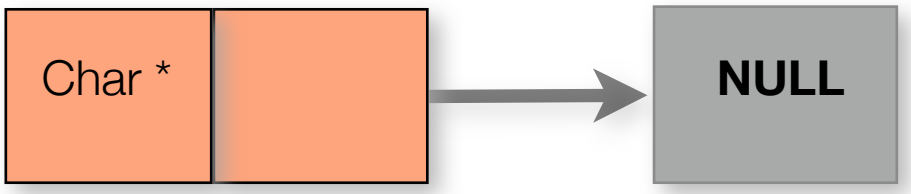
- File non vide
- Le pointeur de fichier qui pointe vers la sortie standard ou un fichier de sortie (c.f extension).

La fonction « write_remove », défile tous les éléments de la file, envoi la chaine sauvegardé sur le flux de sortie, définit encore une fois comme le flux d'entrée (c.f plus haut).
Après utilisation de chaque chaine de caractère défilé, un free.

Le Module File

Mise en place de la File

Nous allons commencer par décrire la structure de notre file, celle-ci est composée de deux pointeurs sur Maillon qui se nomme tête et queue. Ces deux attributs vont donc pointer sur le début (tête) et sur la fin (queue) de notre file. Mais vous vous demandez surement qu'est ce qu'un 'Maillon' ? Pour mieux comprendre un Maillon est ce qui compose notre file. Prenons pour exemple un cas concret comme la file d'attente : La file représentera l'ensemble de notre structure, et chaque personne qui attend dans la file représentera un Maillon de notre file. Et comme dans la file d'attente nous utiliserons le même procédé : premier entré = premier sorti , mais nous détaillerons ceci plus tard dans la programmation de la file. Dernière petite précision avant l'explication graphique, le Maillon lui sera composé de deux attributs qui sont la valeur que contient le Maillon et un pointeur que l'on nommera 'suivant' qui lui pointera sur le Maillon suivant de la file. Passons au dessin :

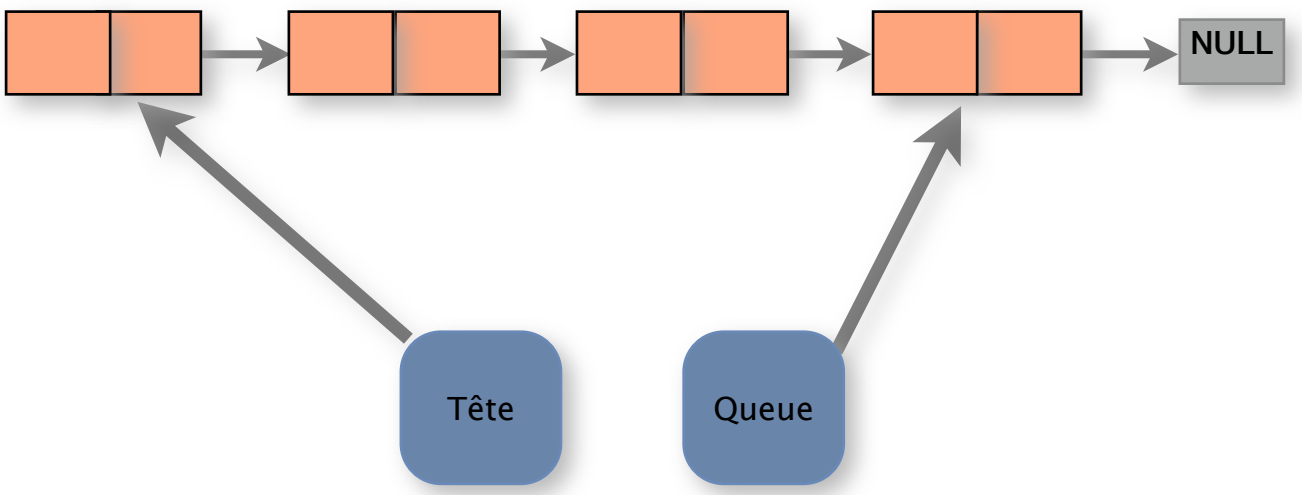


La couleur orange désigne le type Maillon

La couleur grise désigne NULL

Figure 1

La Figure 1 représente donc le type Maillon, nous l'avons adapté à notre projet en spécifiant le type de valeur que contient notre Maillon. Ce sera un pointeur de caractère nommer ligne pour nous permettre de stocker chaque ligne du texte dans un maillon, ainsi pour trier les lignes de notre texte nous aurons qu'a trier les Maillons de la file en fonction de leur attribut ligne.



La couleur bleu désigne le type file

Figure 2

La Figure 2 elle représente la structure file avec ses deux attributs tête et queue qui pointe respectivement sur le début et sur la fin de la file. Et grâce a cette figure on se rend bien compte de notre liste de ligne qui se suit jusqu'à atteindre un Maillon valant la valeur NULL signifiant la fin du texte. Et de plus grâce à cette structure on vois bien que l'on accède facilement au début et à la fin grâce au deux attributs de file et que l'on peut en plus la parcourir grâce à la structure Maillon qui pointe sans cesse sur son suivant.

Manipulation basique de la file

Maintenant que la structure de notre file est défini, nous allons pouvoir nous intéresser à la manipulation de celle-ci et ainsi entrez un peu plus dans le coté programmation.

Initialisation

Commençons tout d'abord par notre fonction d'initialisation, nous la noterons `init ()` dans notre programme. Celle-ci consiste à allouer de la place dans la mémoire, donner une adresse à notre file et de plus mettre la tête et la queue à `NULL` .

Enfilage

Pour enfiler une nouvelle ligne, notre programme prend tout d'abord un chaine de caractère équivalente à la ligne et notre file en paramètre, puis il créer un `Maillon`, lui alloue de la place en mémoire, lui donne une adresse, lui donne comme valeur de ligne la chaine prise en paramètre et fais

pointer son attribut suivant vers `NULL` . Voici l'étape que nous devrons exécuter à chaque enfilage, ensuite vient 3 cas différent :

La chaine en paramètre est `NULL` : on retourne directement zéro.

La file est vide : on fais pointer la tête et la queue vers le `Maillon` que l'on viens de créer précédemment.

La file est non vide : on fais pointer l'attribut suivant de queue sur le `Maillon` que l'on viens de créer et ensuite on fais pointer la queue vers ce même `Maillon`.

Cette fonction renvoie un int qui vaut 1 si l'enfilage a fonctionner et `0` sinon et elle se nommera : `enfiler (char * s, file f)` .

Défilement

Pour le défilement d'une ligne, notre programme prend simplement la file à défiler en paramètre et retourne la ligne qui a été défilé, on la notera : `defiler (file f)` . Il commence à déclarer un pointeur que l'on nommera `m` sur la tête de la file et alloue de la place en mémoire pour une ligne et ensuite 3 cas se sépare :

Le `Maillon` pointant sur la tête est `NULL` : on renvoie alors la valeur `NULL` .

La tête et la queue pointent vers le même `Maillon` (il y a donc 1 seul élément) : on met alors ce maillon à `NULL` .

La file à plusieurs lignes : on fais pointer la tête vers le `Maillon` suivant.

Et ensuite on copie dans une chaine, que l'on a déclarer avant et qui se nomme `c` , la valeur de la ligne de `m` avant de libérer la place en mémoire de `m` et retourner `c` .

Affichage

Pour l'affichage de notre file, notre programme prend en paramètre la file que l'on veut afficher, puis il commence par déclarer un pointeur de `Maillon temp` et deux cas se présente :

La file en paramètre est vide : on affiche que la file est vide.

La file en paramètre est non vide : on fais alors pointer `temp` de la tête jusqu'à la queue en affichant a chaque tour de boucle la valeur de la ligne de `temp`.

La fonction d'affichage ne retourne rien.

Faire le vide

Cette fonction du programme sert à effacer toute trace de la file dans la mémoire. Pour cela on prend en paramètre la file que l'on veut vider, puis on déclare un pointeur de `Maillon temp` qui pointe vers la tête de la file. On libère alors à chaque tour de boucle la place du Maillon puis on le fais pointer sur le `Maillon` suivant jusqu'à ce qu'on est parcouru la file en entier.

Outils avancés de la file

Nous allons voir dans cette partie toutes les fonctions orientées vers le trie de notre file. Elle ont donc été conçue par la demande de mon collègue Soufiane qui a eu besoin, pour réaliser son algorithme `trier_monotonie`, de plusieurs fonctions permettant de manipuler plus en détail la liste.

File vide

La fonction `is_empty (file f)` sert simplement à savoir si la file `f` est vide, pour cela on fais un simple test sur la tête de `f` :

La tête vaut `NULL` : on retourne `1`.

La tête est différente de `NULL` : on retourne `0`.

File triée

La fonction `is_try (file f)` permet de savoir si `f` est trié ou pas, pour cela si la file est non vide alors on définie un pointeur de `Maillon temp` sur la tête de la file qui va parcourir la file tout en comparant la valeur de la ligne de `temp` et de son suivant. Si pendant le parcours une ligne est supérieure à la ligne suivante alors on retourne `0` (la file n'est pas triée) sinon si le parcours a été jusqu'au bout sans rencontrer de problème, alors on retourne `1` (la file est bien triée).

Comparaison

Au niveau de la comparaison nous avons définis deux programmes qui effectue chacun une comparaison sur la file différente. Tout d'abord il y a la comparaison `comparer_tete (file f1, file f2)` qui compare la tête des deux files `f1` et `f2`, et ensuite il y a `comparer_tete_queue (file f1, file f2)` qui elle compare la tête de la file `f1` avec la queue de la file `f2`. Etudions plus en détails ces deux fonctions :

comparer_tete : commence par vérifier si la file est vide, si elle l'est alors on affiche un message d'erreur et l'on quitte prématurément le programme, sinon on retourne alors la comparaison des deux chaines de caractère contenue dans le `Maillon` tête des deux files. Cette comparaison est effectuée grâce à la fonction `strcmp (char * chaine1, char * chaine2)` qui est issue de la librairie `<string.h>` et qui retourne `0` si les chaine 1 et 2 sont égales, un nombre positif si chaine1 est supérieur à chaine2, et un nombre négatif sinon.

comparer_tete_queue : qui réalise exactement le même procédé que la fonction précédente sauf qu'au niveau de la comparaison retourner elle est effectué entre la ligne du `Maillon` tête de `f1` et la ligne du `Maillon` queue de `f2`.

Transfert

La fonction transfert `(file f1, file f2)` réalise, comme son nom l'indique, un transfert de ligne entre la file `f1` et la file `f2`. Pour cela nous déclarons un pointeur de caractère `c` nous permettant de définir une ligne, puis nous défilons la tête de `f1` dans `c` grâce à la fonction `defiler (file f)` vu précédemment. Ensuite on vérifie la valeur du pointeur `c`, si celui-ci vaut `NULL` alors

on retourne `0`, sinon on retourne un `int` de validation qui nous sera retourner par la fonction `enfiler(char * s, file f)` que nous utiliserons avec comme paramètre :

`char * s` : Le pointeur de caractère `c` contenant la ligne de tête de `f1`

`file f` : Ce sera la file `f2` contenue en paramètre de `transfert` .

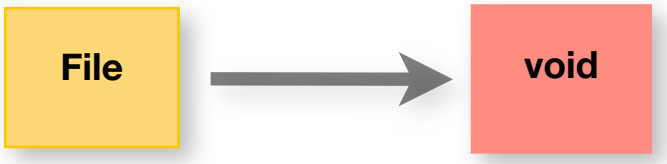
Ainsi la file `f1` aura été défiler et la file `f2` aura été enfilé avec la tête de `f1`, il y a donc eu un transfert de ligne de `f1` à `f2`. De plus cette fonction nous retournera `0` si le transfert c'est bien exécuté et `1` sinon.

Nombre de lignes

La fonction `there_is (file f, int i)` permet de savoir s'il y a tant de lignes dans la file ou non. Pour cela cette fonction utilise d'abord deux paramètre qui sont la file de lignes qui nous intéresse et un entier `i` qui spécifie le nombre de lignes dont on cherche l'existence dans la file. Puis on définis un pointeur de `Maillon` sur la tête de la file (comme dans les cas précédents ce pointeur de `Maillon` nous servira à parcourir la file sans la modifier) et on parcours `i` `Maillon` de la liste. Si la boucle arrive à son terme alors on retourne `1` pour exprimer qu'il existe bien `i` ligne dans la liste, sinon cela veut dire que l'on a rencontrer un `Maillon` valant `NULL`, et donc qu'il y a moins de `i` lignes dans la file, on retournera alors `0` pour l'exprimer.

Masquage de la file

Nous allons maintenant parler du masquage de notre file, pour cela nous utilisons le type `void *` qui sera le type de notre file. Notre ancien type file sera donc remplacé par la structure `Chaine` qui sera équivalente à l'ancien type file (voir Figure 2). Cela veut donc dire que quand l'on utilisera notre file en dehors du module file elle ne sera vue que par un pointeur qui pointe vers `void`, ainsi on aura accès ni au `Maillon` tête ni au `Maillon` queue et donc a aucun élément de notre structure. C'est donc pour cela que toutes nos manipulations de file ne se font que par des paramètre et des retours de type file ou de type connu par tous. Et ainsi chaque fonction de notre module file ayant besoin de faire appel au type file, font d'abord une définition de pointeur de `Chaine` et «cast» la file à manipuler dans cette `Chaine`. C'est cette opération que l'on retrouve souvent au début des fonctions : `Chaine * cop = (Chaine *) f` . On utilise donc tout simplement `cop` pour manipuler la file `f`.



La couleur jaune désigne le nouveau type file

Figure 3

Cette Figure 3 nous montre comment notre type file est perçu en dehors de notre module file. Comme dis précédemment la seule information que l'on a et qu'elle pointe vers un type void qui est donc équivalent à n'importe quel type.

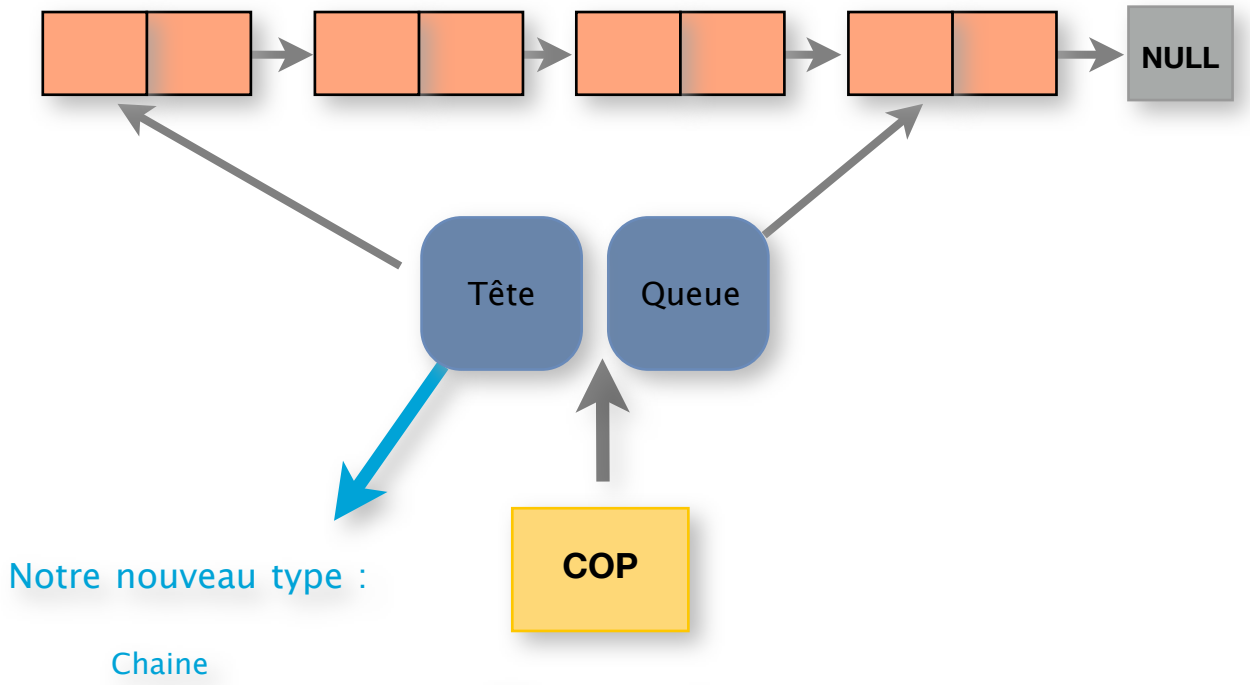


Figure 4

La Figure 4 elle représente aussi notre file mais après avoir fais le «cast» vu précédemment. Ce sera donc le pointeur de `Chaine cop` qui pourra avoir accès à tous les élément de notre file, on est passé du type `void *` au type `Chaine *` .

Sort

Introduction

Je voudrai d'abord commencer par un petit rappel de l'algorithme de tri monotonie, que nous avions confondu (notre groupe), à l'algorithme de tri par fusion, deux algorithmes différents, mais qui partent du même principe qui consiste à diviser pour régner.

bref le principe est simple :

- Si on trouve une monotonie
    - on coupe la fonction en séparant les monotonie suivis dans deux coupes différentes.
    - on fusionne tout en respectant un ordre de tri choisis.

et on rappel toujours la même fonction jusqu'à ce qu'on ai plus de monotonies.

N.B :

Nous avons mis en place deux macros l'une est « EG_ZERO (adr) », qui vérifie si une adresse est différente de NULL si c'est le cas on affiche un message d'erreur et on quitte, l'autre est ECHEC(x, message), qui vérifie le code de retour « x « et affiche «message» comme message d'erreur, et quitte le programme.

La fonction « couper »

Rappel : La fonction « couper », comme son nom l'indique, coupe la file de départ dans les deux files passé en paramètre de celle-ci, donc par ce fait, sépare les monotonie «collé» dans les deux coupes, (c.f schéma 1)

soit la file suivante : (à commencer par la gauche)

5	3	2	1	10

Soit la coupe1 en rouge, et la coupe2 en noire. les monotonie ayant la même couleur doive être enfiler dans la même coupe.

Pour se faire nous avons utilisé un algorithme récursif, simple et compressé :

Entrée : la file qui doit être coupé, la file contenant la première coupe, la file contenant la deuxième coupe.

Cas de base :

- si la file est vide on a finit.

Sinon :

- on regarde si les deux files, coupe1 et coupe2 sont vide, si c'est le cas, alors c'est la première fois qu'on effectue l'opération de coupe, et donc on choisis une des deux files pour défiler et enfiler dedans. et on rappel la fonction. un point très important dans cet algorithme, est que l'ordre des 3 paramètre est très important, car le premier est celui de la file d'origine, mais le deuxième paramètre et sensé être la coupe dans la quelle on a enfiler le dernier élément enfilé !! donc quand on va rappeler la fonction il faut toujours mettre dans le deuxième emplacement la coupe qui a reçu l'élément, dans ce cas là, la coupe 1

- si l'une des deux coupe n'est pas vide, on compare la tête de la file d'origine, avec la queue du deuxième paramètre qui est la «coupe1» (change en cas d'appel avec couper (file, coupe2, coupe1)) en utilisant la fonction «comparer_tete_queue»
  - si on trouve une monotonie on transfert l'élément vers l'autre coupe (3eme paramètre) et rappel la fonction
  - sinon on transfert l'élément vers la coupe courante.

La fonction « fusionner »

Cette fonction permet de fusionner deux files, en une seule file, tout en triant les éléments.

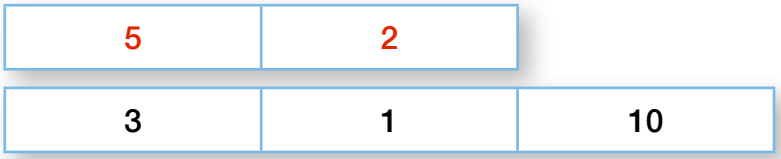
Entrées : coupe 1, coupe 2, File de résultat de fusion

L'algorithme de fusion :

- tant que l'une des deux files n'est pas vide
  - on compare les deux têtes de files des deux coupes et on enfile le plus petit (si on trie par ordre croissant) dans la file de résultat
- on vérifie si les deux files sont bien vide, sinon on enfile le reste de la file qui n'a pas été encore vidé.

Exemple :

continuons l'exemple de tout à l'heure, et donc après coupe on a obtient les deux coupes suivantes



Après l'emploie de la fonction «fusionner» nous allons obtenir la chose suivante :

La fonction « sort »

La fonction «sort» prend en paramètre la file-à-trier, et utilise les deux sous-programmes définit plus haut pour «couper» et «souder» et cela d'une manière répétitive, tant qu'on le programme trouve au moins une monotonie.

Dans les sous programmes «couper» «fusionner», nous avons utilisé que des pointeurs universels pour éviter d'utiliser trop de «malloc» dans le programme, et sur le coup se perdre dans son propre programme et risquer des fuites de mémoires (du moins limiter ...), c'est pour cela que la fonction sort s'occupe de créer les deux files de coupes, appelle les deux fonctions «couper» et «fusionner», ensuite libère ces deux coupe juste avant chaque appelle récursif, la condition pour pénétrer dans le coeur de l'algorithme étant d'avoir au moins une monotonie.

Extension

Option ligne de commande

Faute de temps nous avons pu faire que l'extension qui consiste à prendre en charge des paramètres au niveau du programme.

Principe des paramètres « -r » & « -u »

Nous avons définit deux variables de type entier : «inverse» et «no_repeat»

Comme leurs noms l'indique l'une pour activer le tri à l'ordre inverse (paramètre -r), et l'autre pour supprimer les occurrences (option -u)

Ces deux variable sont mis à 1 lorsque les options correspondant sont détecter à l'entrée, les paramètre étant prioritaire dans le programme.

Activation de commande « -r » & « -u »

La vérification des commandes se fait avant la lecture de l'entrée standard, c'est à dire qu'on vérifie d'abord si on a des arguments, et en fonction de ces argument on a affecte à des variables les valeurs correspondante à l'option saisie.

Redirection à partir de paramètre

Redirection de sortie standard avec un paramètre «nom_fichier»

Après la vérification des commandes « -r » et « -u », viens la vérification de la redirection de la sortie standard, on vérifie si en premier la fonction «-o» a été activer, le nom fichier au quel on associera la sortie standard sera inscrit après la fonction bien évidement.

- Variable : FILE * fich_sortie qu'on associe au départ à la sortie standard (par défaut).

Redirection d'entrée standard avec un paramètre «nom fichier»

Même principe que pour la redirection de sortie standard.

- Variable : FILE * fich_entrée qu'on associe au départ à l'entrée standard (par défaut).

Changement au niveau du tri_monotonie

Tri inverse avec la commande « -r »

Pour exploiter ces deux information, nous remarquons que pour trier d'une manière inverse, il faut détecter les monotonies d'une manière, inverse à la manière habituelle (ordre croissant), ce qui implique un changement dans la fonction «coupe», ensuite il faut trier d'une manière inverse, donc le programme «fusionner» changera un petit peu aussi.

Pour répondre à se changement, sans changer (de trop) le programme, ni ajouter des nouvelles fonctions, il suffit d'ajouter une condition à la ligne qui correspond à la comparaison effectué dans le sous programme «couper», et le sous programme «fusionner» :

donc : si le tri inverse est activé le résultat de comparaison est multiplier par (-1), sinon on laisse le résultat comme avant.

Le traitement du résultat de comparaison reste le même.

Suppression d'occurrences avec la commande « -u »

Pas la peine de courir après les occurrences et les chercher partout ... nous savons que le programme de tri va nous les réunir lui même donc :

Le mieux est de le faire pendant le tri, c'est à dire quand on fusionne.
Il suffit d'ajouter une condition juste avant de transférer l'élément vers la file d'origine, on compare cet élément avec le dernier éléments qui a été transféré dans la file d'origine, s'ils sont égaux, alors on supprime l'élément sinon on l'enfile.

Option Ligne de taille quelconque

Une fois la partie principale terminée, nous nous sommes lancés dans la réalisation de l'extension « lignes de tailles quelconques.

Le projet principal permettait à l'utilisateur d'entrer des lignes de 256 caractères, puis de les « enfiler » dans notre file. Nous devions maintenant supprimer cette limite qui imposait à l'utilisateur d'entrer autant de caractères qu'il souhaitait.

Le premier problème que nous avions à résoudre était le suivant : comment faire pour stocker des lignes sans en connaitre la taille ? En effet, avant de créer une chaine de caractère, nous devions allouer la mémoire nécessaire au stockage de celle-ci, nous ne pouvions donc pas tout d'abord allouer la mémoire nécessaire, puis entrer la chaine de caractère.

Nous avons donc recherché une manière différente permettant de récupérer les chaines de caractères. Nous avons alors pensé à utiliser la fonction fgets (une alternative de la fonction scanf), celle-ci fonctionne de la manière suivante : elle utilise trois arguments :

- Un pointeur vers un tableau de caractères.
- La taille du tableau que nous voulons créer. •
- Un pointeur sur le fichier que nous allons lire : étant donné que nous

récupérons les chaine de caractères sur l'entrée standart, nous indiquerons « stdin ».

Cette fonction renvoie pour finir, un pointeur vers le tableau de caractères dans lequel nous venons d'écrire.

Grace à la fonction fgets, nous pouvions donc récupérer une chaine de caractères et savoir s'il reste encore du texte à récupérer, en effet, lorsque l'utilisateur entre sa chaine de caractères, il termine obligatoirement par le caractères « entrée » ; de plus, contrairement à la fonction scanf, lorsque l'utilisateur entre une chaine de caractères plus longue que celle qui est imposée, les caractères qui ne sont pas pris restent dans une partie appelée « buffer », on peut donc, grâce à la fonction fgets, récupérer les caractères manquants.

La fonction fgets va donc récupérer la chaine de caractères dans « le buffer », celui-ci est un intermédiaire entre le clavier et la fonction fgets : lorsque l'utilisateur entre une chaine de caractère, celle-ci est enregistrée dans le « buffer », la fonction fgets va donc ensuite récupérer dans le buffer la chaine de caractère. Nous pouvons donc utiliser plusieurs fois la fonction fgets jusqu'à ce que le buffer soit vide (on peut donc récupérer une chaine de caractère de taille quelconque, même si celle-ci sera en plusieurs morceaux).Nous avions cependant un nouveau problème : comment savoir lorsque le buffer est vide ?

Après quelques recherches concernant la fonction fgets, nous avons découvert que celle-ci enregistrait également le caractère « entrée » : lorsque l'utilisateur entre une chaine de caractères, celui-ci appuie sur la touche « entrée » lorsqu'il a fini et donc, nous n'avons plus de caractères à récupérer dans le « buffer ». Nous avons donc commencer par réaliser l'algorithme suivant afin de pouvoir ensuite réaliser le programme en langage C qui va gérer cette extension.

Algorithme ligne quelconque :

- Début.
- Tableau de caractères chaine ;
- Variable file de type File ;
- Initialisation de la file ;
- Afficher « Veuillez entrer une chaine de caractères » ;
- Tant que la chaine de caractères ne contient pas le caractère « Entrée »faire
- Récupérer la chaine de caractères grâce à la fonction fgets ;
- Enfilage de la chaine dans la file ;
- Fin tant que
- Affichage file
- Fin

Le principe de l'algorithme est le suivant :

- On créer une chaine de caractères pouvant contenir un nombre précis de caractères, une variable de type File.
- On initialise la file.
- Tant que la chaine de caractère ne contient pas de caractère « Entrée » : on récupère une nouvelle chaine de caractères pour ensuite l'enfiler dans notre file.

Faute de temps nous n'avons pas pu intégrer cette fonctionnalité au projet, le code source réalisé par Aflah Sofiane est en annexe.

Test, problématique ...

Valgrind

Nous avons eu du mal a utilisé valgrind, au long de la programmation de l'application. puisque nous travaillons sous Mac OS X.

Mais j'ai finalement pu l'utiliser est réparé la plupart des fuites mémoires, hélas il en reste une que j'ai pas pu trouvé, au niveau de la fonction défiler, malgré le fait qu'on supprime le maillon qu'on défile, après récupération de l'information contenu de ce dernier, le problème reste présent. (je vous ai envoyé un mail de secours)

Makefile

Nous avons compris le sujet d'une manière assez différente au début, nottamment avec la gestion des flux. Mais après réflexion nous avons compris le fonctionnement du makefile, et avons su l'exploiter.

Seulement pour le test du makefile en utilisant le fichier «trier.c» nous avons eu des résultats un peu inattendu, car en utilisant un fichier texte par exemple «makefile.txt» le test marche bien.
Après comparaison à la main des deux fichier retourné par l'application d'un coté et la fonction sort de l'autre, nous remarquons que la différence est minime et quelle se limite à quelques espaces près ou tabulation.

Répartition des tâches

Faute de niveau et de connaissances différentes, de la formation que Monsieur Aflah sofiane a suivis, il n'avait les bases requise (notamment le principe de fonctionnement des maillon et listes chainées), donc une partie du groupe a commencé en avance pendant que l'autre partie rattrapait son retard.

Répartition des tâches :

Aflah Sofiane :

- Remise à niveau en fonctionnement de structure chaîné, et bases de programmation «objet»
- Re-codage (pour entrainement) de la file simple (enfiler - défiler - afficher)
- Codage d'extension «lecture de ligne infinie» (non mis au point)

Hubert Simon :

- Codage du Module File.c
- Extension : Option en ligne de commande
- Gestion de flux

Boulealf Soufiane :

- Algorithme de tri monotonie
- Adaptation de l'extension au module de tri
- Gestion de Mémoire

Annexe

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAILLE 7
#define FIN "EOF"


typedef struct Maillon Maillon;
struct Maillon
{
    char *ligne;
    Maillon *suivant;
};

typedef struct File File;
struct File
{
    Maillon *tete;
    Maillon *queue;
};


void init(File *file)
{
    Maillon *maillon;
    maillon = malloc(sizeof(Maillon));
    maillon->ligne = NULL;
    maillon->suivant = NULL;
    file->tete = maillon;
    file->queue = maillon;
}


void enfilage (File *file, char *chaine)
{
        if ((file->tete)->ligne == NULL)
        {
            (file->tete)->ligne = chaine;
        }
        else
        {
```

```
            Maillon *nouvmaillon;
            nouvmaillon = malloc(sizeof(Maillon));
            nouvmaillon->ligne = chaine;
            nouvmaillon->suivant = NULL;
            (file->queue)->suivant = nouvmaillon;
            file->queue = nouvmaillon;
        }
}

void affichage (File *file)
{
    Maillon *afficheur;
    afficheur = file->tete;
    if(afficheur == NULL){
        printf("vide vide ");
        exit(0);
    }
    do
    {
        printf ("%s", (afficheur->ligne));
        afficheur = afficheur->suivant;
    }while (afficheur!= NULL);

}

int main(void)
{
    File *file;
    char *chaine;
    file = malloc(sizeof(File));
    init(file);
    //chaine = malloc(TAILLE*sizeof(char));

    printf ("Entrez une chaine de caractere\n");

    do
    {
        chaine = malloc(TAILLE*sizeof(char));
        fgets(chaine, TAILLE, stdin);
        enfilage(file, chaine);
        printf("%s", chaine);

        while (strchr(chaine, '\n') == NULL)
        {
            chaine = malloc(TAILLE*sizeof(char));
            fgets(chaine, TAILLE, stdin);

            enfilage(file, chaine);
            printf("%s", chaine);
```

```
            printf("Enfilage\n");
        }
        printf("Sortie premiere boucle\n");
    } while (strcmp (chaine, FIN) != 0);



    affichage(file);


    return 0;
}
```